COMS W4160— Computer Graphics Spring 2014

Programming Assignment 2

out: Feb 25, Tuesday
due: Mar 13, 11:59PM Thursday (submit to courseworks)

In this programming assignment, you will animate and pose 3D articulated characters by implementing forward and inverse kinematic methods as discussed in class. You should be able to animate character joint angles to demonstrate character moving (forward kinematics), as well as pose the character by dragging the points on screen and moving the skeletons (inverse kinematics).

1 Getting Started

The mathematics of forward and inverse kinematics were discussed in lectures, and related supplemental materials can be found on coursewiki. For starter code, you can use that of the first programming assignment or a stripped-down version of your first programming assignment submission. You are free to extend or modify your first programming assignment code and use any 3rd-party library to help you solve the linear systems. Later in this handout, we recommend a few libraries you might find useful.

2 Programming Requirements

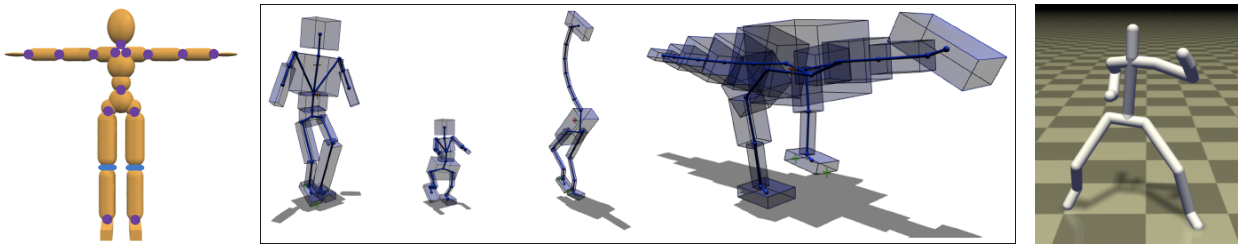
This assignment has the following steps:

2.1 Geometric and Kinematic Modeling

You need only use simple geometric representations for your 3D "characters". For example, you can make simple models using colored boxes, ellipsoids, cylinders, spheres, etc. Figure 1 shows a few examples using simple primitives. If you would like to make it fancier, you can model your own character parts in 3D software (such as Maya and Blender) or import them from Turbo Squid.

The first step is to model each rigid-body part (bones) of the character in a rest pose ($\theta_j = 0, j = 1 \dots n$). You start by defining each rigid link's geometry, then you can define each join's center position, $\boldsymbol{c}_j$, and the rotation axis $\boldsymbol{r}_j$. **Only *revolute* joints are required in this assignment**; other types of joints are optional. In your code, you also need to build a tree structure of the kinematic model to maintain the parent/child relationships among those rigid links. These relationships will be used to construct the transformation matrix to transform from each link's local frame of reference into the world frame of reference. Since we assume that each joint is located at the origin in each link's own local frame, you can translate each link's geometry so that its joint center is at the origin of its local body coordinate frame. You are free to do this modeling in whatever manner you feel comfortable, e.g., hard-coded numbers, text configuration files, via a custom mouse interface in "edit mode", or a separate program, etc.

What to model: You are required to build two character models, and more are optional: (1) a planner serial-chain manipulator with at least 3 links, and (2) either a model of your hand or an articulated figure of your choosing. There is no limit on how many examples you provide. To debug your implementation, we suggest you start from simple planar manipulators (e.g. ones with 1 link, 2 lines, and 3 links).



(a) [LYvdP+10] (b) [dLMH10] (c) [LWH+12]

Figure 1: **Simplified character geometry** is sufficient for the assignment, and is commonly used in computer graphics for proof-of-concept demonstration of character animation techniques.

2.2 Forward Kinematics

Once you have a tree-structured model defined you can start animating it using forward kinematics. Your program should enable a user to interactively adjust the joint angles, $\{\theta_i\}$, by either dragging a slider or pressing some keys, and meanwhile, you should update the body-to-world transformation matrices (as we discussed in class), then draw the articulated geometry.

2.3 Inverse Kinematics

As discussed in class, you should implement an inverse kinematics solver based on a damped least-squares solver (a.k.a. regularized least-squares, or ridge regression). You can also find details about damped LS method in the IK tutorial on coursewiki or [DW95]. The key part of your implementation is to solve least-square problems of the form,

$$\boldsymbol{J}\Delta\boldsymbol{\theta} = \Delta\boldsymbol{p},$$

where $\Delta\boldsymbol{\theta}$ are the changes of N joint angles, $\Delta\boldsymbol{p}$ are the changes in m constraint positions (e.g. the link positions specified by mouse dragging) in the world frame of reference, and $\boldsymbol{J} = \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{\theta}} \in R^{M\times N}$ is the Jacobian matrix. Depending on the number of position constraints, you may end up with an onver-determined ($M > N$) or under-determined ($M < N$) system. The damped least-square solver will generate angle updates of the form

$$\Delta\theta = (\boldsymbol{J}^T\boldsymbol{J} + \delta\boldsymbol{I})^{-1}\boldsymbol{J}^T\Delta\boldsymbol{p},$$

where $\delta > 0$ is a user-specified regularization parameter to avoid singular matrices. Note that for small values of δ (e.g., $\delta < 10^{-6}\|\boldsymbol{J}^T\boldsymbol{J}\|_F$), you will definitely need double-precision calculations. You need to try different δ values to give the best results.

2.3.1 User Interface Requirements

To help evaluate your submission, please ensure your application supports the following user interfaces:

- **Spacebar toggles between forward and inverse kinematics:** In forward-kinematics mode, the model can be animated by changing joint parameters. As a bonus (see below), you can load keyframes and play it back. In inverse-kinematics mode, the model only moves when the user clicks-and-drags on the object and adjust the joint angles.

- **Keys to select the kinematic model:** If you build up the two kinematic models in your code, you should provide keys to select among them. For example, "1" might select a 2D serial-chain example with two links, "2" for a chain with 10 links, "3" a hand model, etc.. If your code is to load a kinematic model from a configuration file, you do not need to provide this interface.

- **Mouse-based character posing:** You will implement a mouse-based interface to interactively and intuitively pose your character while in "inverse-kinematics mode". You should be able to add/remove position (pin) constraint on your character, and drage these constraints around to poise it. Specifically,

  – *Picking* should be implemented so that given a mouse click you can determine which, if any, link has been selected and at what position. See the starter code for a demonstration of mouse selection. By pressing some keys, a new constraints should be added and drawn on the screen.

  – *Dragging* should allow you to move constraints around in the plane perpendicular to the view direction–the model-view matrix will help you find this direction and plane. In this way, it should be intuitive to fix a body in space, such as the feet, and move other parts around, e.g., to make a character wave.

  – *IK solution* should proceed in an incremental manner by updating the joints a small amount at each time step, which involves recomputing and solving the Jocobian LS system each time.

2.4 Bonus Points

We highly encourage creative (and even crazy) work! There are a few points you might consider to earn bonus points. Of course, you are not limited to those ideas, and anything creative will give you bonus points.

- **extending to other types of joints:** Beside robolute joints, you might want to consider other types of joints used in your characters, including prismatic joints, cylindrical joints, spherical joints, etc. They can fit into the code with the same derivation as we did for revolute joints but with different DoFs.

- **mesh skinning:** As we discussed in class, you can attach a triangle mesh on the skeleton represented by rigid links. The triangle mesh can be deformed using Linear Blending Skinning, which update the position of a vertex i from its undeformed position $\boldsymbol{p}_i$ via

$$\boldsymbol{p}_i' = \sum_{b=0}^{|\mathcal{B}|} w_{ib} \boldsymbol{T}_b \boldsymbol{p}_i,$$

  where the per-vertex bone weights, w_{ib}, typically only weight to a small number of bones (or links). In practice, non-negative weights are chosen that weight vertices heavily to their nearby bones. These weights can be specified using tools like Blender and exported into a text file for your program to load.

- **key-frame animation:** Use your IK implementation to produce keyframes of the joint parameters, then play them back by interpolating key-framed joint angles using spline curves, which we will discuss soon in class. Dump frames of your creation to make a cool video highlight.

3 Submission and FAQ

Submission Checklist: Submit your assignment as a zip file via courseworks. Your submission must consist of the following parts:

1. **Documented code:** Include all of your code and libraries, with usage instructions. Your code should be reasonably documented and be readable/understandable by the TAs. If the TAs are not able to run and use your program, they will be unable to grade it. Try to avoid using obscure packages or OS-dependent libraries, especially for C++ implementations. To ensure the TAs can grade your assignment, it is highly suggested to compile your code on CLIC machines, and include details of compiling and running your code on CLIC.

   In the starter code, we also include a CMakeLists.txt file which is used to automatically detect libraries and generate Makefile using the cmake building system. It is optional to modify the CMake file for your project, while it will probably make the compile more organized and less tedious.

2. **Brief report:** Include a description of what youve attempt, special features youve implemented, and any instructions on how to run/use your program. In compliance with Columbia's Code of Academic Integrity, please include references to any external sources or discussions that were used to achieve your results.

3. **Video highlight!:** Include a video that highlights what youve achieved. The video footage should be in a resolution of 960×540, and be no longer than 10 seconds. We will concatenate some of the class videos together to highlight some of your work.

4. **Additional results (optional):** Please include additional information, pictures, videos, that you believe help the TAs understand what you have achieved.

Evaluation: Your work will be evaluated on how well you demonstrate proficiency with the requested forward/inverse kinematics functionality, and the quality of the submitted code and documentation, but also largely on how interesting and/or creative your overall project is.

A Numerical Libraries

You are free to use any numerical libraries to help you solve the least-square problem. Please describe in your report what library your program uses and how we should install it on a local directory when we grade it on CLIC machines. We recommend two libraries that can be useful for your program. Both of them provides the least-square solves. So you can choose to use either of them.

- **the Eigen Library** is a C++ template library for linear algebra. You don't need to install it, instead you simply put it on a folder and include that folder in your code. The online documentation provides detailed examples including various least-square solves and other numerical methods. It is **highly** recommended.

- **the GNU Scientific Library** is another useful numerical library, which provide a wide range of method for scientific computing, and linear algebraic methods are part of it. You need to install it or compile it on your machine, and link your program against its libraries. For PA-2, we suggest you prefer Eigen over GSL.

B C++ starter code

You can use the C++ starter code from PA-1. Again, we attach the instructions of compiling it here. The starter code is based on glut. On Linux or Mac OS, you can compile the starter code by the following command

```
unzip pa1_starter.zip && cd pa1_starter && make
```

As you add your implementation files, you need to modify the Makefile to compile your code. To make it easier, we also include the configuration file for using cmake, a cross-platform building tool, to build the code. On Ubuntu, you can install cmake by the command

```
sudo apt-get install cmake
```

The following command sequence will make code compiled on both Linux and MacOS.

1. `unzip pa1_starter.zip`
2. `cd pa1_starter && mkdir gcc-build && cd gcc-build`
3. `cmake ..`
4. `make`

Library dependence: The starter code depends only on the OpenGL library and GLUT. Both of them have been installed on CLIC machines. Most of the operation systems with graphics interface should have them already. If not, they can be manually installed. On Ubuntu Linux, you can install them by

```
sudo apt-get install libglu1-mesa-dev freeglut3-dev
```

References

[dLMH10] Martin de Lasa, Igor Mordatch, and Aaron Hertzmann. Feature-Based Locomotion Controllers. *ACM Transactions on Graphics*, 29(3), 2010.

[DW95] Arati S Deo and Ian D Walker. Overview of damped least-squares methods for inverse kinematics of robot manipulators. *Journal of Intelligent Robotic Systems*, 14(1):43–68, 1995.

[LWH+12] Sergey Levine, Jack M. Wang, Alexis Haraux, Zoran Popović, and Vladlen Koltun. Continuous character control with low-dimensional embeddings. *ACM Transactions on Graphics*, 31(4):28, 2012.

[LYvdP+10] Libin Liu, KangKang Yin, Michiel van de Panne, Tianjia Shao, and Weiwei Xu. Sampling-based contact-rich motion control. *ACM Trans. Graph.*, 29(4):128:1–128:10, July 2010.